TECHNICAL REPORT 1956
April 2007

Composeable Chat over Low-Bandwidth Intermittent Communication Links

D. R. Wilcox

SSC San Diego
San Diego, CA 92152-5001

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
APR 2007		**00-00-2007 to 00-00-2007**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Composeable Chat over Low-Bandwidth Intermittent Communication Links	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Rand Corporation,1776 Main Street,PO Box 2138,Santa Monica,CA,90407-2138**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**40**	

EXECUTIVE SUMMARY

PROBLEM

Intermittent low-bandwidth communication environments, such as those encountered in U.S. Navy tactical radio and satellite links, have special requirements that do not pertain to commercial applications. They need a bandwidth-compression algorithm that limits the dependence of encoding symbols on previous state information because the loss of the previous state has a ripple effect on the interpretation of what follows. They also need a homing mechanism to permit synchronization at an arbitrary point in the broadcast message stream when reception is re-established, ideally, without negotiation between the transmitter and the receiver.

APPROACH

Small Text Compression (STC), introduced in this report, is a data compression algorithm intended to compress alphanumeric text by encoding it using 5-bit characters.

STC supports synchronization at message boundaries using a uniquely identifiable 10-bit homing sequence. Compression state is not propagated across message boundaries. Messages are suitable atomic units of synchronization because partial corruption within a given message generally invalidates the entire message in receiving applications.

The 5-bit character encoding of STC can be extended to encompass schema-based XML compression. This approach replaces XML element and attribute tags with integer constants the value of which have been predefined by traversing the applicable XML schema and enumerating its relevant nodes. Because the transmitter and the receiver already have identical knowledge of the mapping between integers and tags, that knowledge is not transmitted within the message.

Within the bounds of an atomic message, further compression can be obtained by noting that fewer integer bits are required to map the children of a parent XML element when the identity of the parent element is known. An escape mechanism is provided for cases where a single 5-bit character is insufficient.

CONCLUSIONS

STC compresses 8-bit ASCII English text to approximately two-thirds of its original size regardless of text length. STC is superior to gzip and zlib for text lengths less than 100 characters, typical of chat user content, and is competitive in the 200- to 300-character range.

The degree of XML schema-based compression depends on the verboseness of the original XML tags, and to some extent, the flatness of the hierarchy defined by the schema, both of which are application dependent.

Overall compression of short TCP/IP messages is still limited by the TCP/IP envelope itself, which cannot be compressed and remain interoperable.

CONTENTS

EXECUTIVE SUMMARY ..... **iii**

1. INTRODUCTION **1**

1.1 COLLABORATION DEFINITIONS 1
1.2 INTERMITTENT LOW-BANDWIDTH CHAT 1
1.3 REPORT ORGANIZATION 2

2. CHAT SERVER ARCHITECTURE **3**

2.1 CHAT SERVER ANATOMY 3
2.1.1 Crossbar Switch 3
2.1.2 Message Management 8
2.1.3 Message Distribution 11
2.2 NETWORK CONNECTIVITY 12

3. DATA COMPRESSION **13**

3.1 COMPRESSION TECHNIQUES 13
3.1.1 Symbol Compression 13
3.1.2 Sequence Compression 14
3.1.3 Context Compression 16
3.2 INTERMITTENT CONNECTIONS 16
3.2.1 State Containment 17
3.2.2 State Synchronization 17
3.2.3 State Codebook 17
3.3 SMALL TEXT COMPRESSION 18
3.3.1 Compression Design 18
3.3.2 Compression Implementation 20
3.3.3 Synchronization Design 20
3.3.4 Synchronization Implementation 21
3.3.5 Performance 23
3.3.6 Limitations 25
3.4 XML-SCHEMA-BASED COMPRESSION 26
3.4.1 Name Tag Translation 26
3.4.2 Contextual Name Tag Translation 27
3.4.3 State Synchronization 29
3.4.4 State Syntax Checking 30
3.4.5 User Data 31
3.5 SMALL TEXT XML COMPRESSION 31
3.5.1 Name Tag Integer Subset 31
3.5.2 Name Tag Integer Separation 32
3.5.3 Name Tag Integer Digits 32

4. BIBLIOGRAPHY **35**

Figures

1. User channel join diagram 3
2. Links to user node 6
3. Segments of messages held in separate buffers during transactions 9
4. Message as a vector of message-segment links 10
5. Example of the use of “1” and “0” as most significant bit in UTF-8 14
6. Example of the use of “1” and “0” as most significant bit in WBXML 14
7. Tree structure for storing words in a text file 15
8. Compression algorithm comparison 24
9. Effect of zlib 25
10. Large-value name tag integer example 33

Tables

1. Characteristics of the four types of collaboration 1
2. Compressed 5-bit character codes 19
3. Compression example 20
4. 5-bit character synchronizer state machine 22
5. 5-bit character reader state machine 23

1. INTRODUCTION

1.1 COLLABORATION DEFINITIONS

Computer-based communication among two or more human users is called **collaboration**. Four basic types of collaboration employ primarily alphanumeric text: email, wiki, instant messaging, and chat (Table 1). These types can be differentiated by two properties.

The first property is whether they require users to be connected to the same collaboration environment at the same time to communicate. A user who is connected to a specific collaboration environment is said to be **present** within that environment. Collaboration that requires the users to be present simultaneously is called **synchronous collaboration**. Collaboration that does not require their simultaneous presence is called **asynchronous collaboration**. Asynchronous collaboration systems store messages to make them accessible when their recipients become available.

The second property is whether user messages are addressed directly to users or addressed to meeting places. In the latter case, recipients, if there are any, receive messages by being present at the addressed meeting place. The distinction is analogous to the distinction between unicast and multicast in a local area network. Unicast messages directly address a specific network interface. Multicast messages, on the other hand, address a virtual port to which a number of network interfaces may be listening.

Table 1. Characteristics of the four types of collaboration.

	Targeted Users Do Not Need to be Present to Receive Messages (Asynchronous)	Targeted Users Need to be Present to Receive Messages (Synchronous)
Messages Addressed Directly to Users (Unicast)	Email	Instant messaging
Messages Addressed to a Meeting Place (Multicast)	Wiki	Chat

Text chat, the subject of this report, is near-real-time synchronous collaboration addressing alphanumeric messages to a common meeting place, called a **channel** or **chat room**. Some chat servers support message logging. The ability to save messages for later retrieval endows these servers with asynchronous properties normally associated with email or wiki. Chat servers usually also implement instant messaging within a unified protocol by allowing messages to address both specific users and chat channels. All this can be a source of confusion when a chat server implements more than chat. It is important to keep the four collaboration concepts distinct because their implementation algorithms are distinct.

1.2 INTERMITTENT LOW-BANDWIDTH CHAT

In a traditional chat system, users connect to a centralized chat server over a network such as the Internet. Currently available off-the-shelf commercial products assume the existence of a reliable high-bandwidth network. In the U.S. Navy tactical environment, because of limitations in the radio-frequency spectrum and platform dynamics, many units are limited to radio and satellite communication links through intermittent low-bandwidth connections. The performance of these links is similar to a bad connection over a dial-up modem. This report describes initial research aimed at mitigating these limitations.

The research focuses on addressing the following navy tactical user needs. The low-bandwidth environment necessitates identifying streamlined message protocols and data compression. The intermittent-connection environment also necessitates an efficient reconnection mechanism that requires little or no user interaction. The user must be able to view time-stamped past message content to see specifically what is missed during a connection interruption and to gain context when replacing the previous user during a user shift change. Finally, the user must be able to obtain feedback that critical messages sent to recipients have been received.

1.3 REPORT ORGANIZATION

This report is divided into two major sections.

The first section examines the implementation of a chat server. It reflects experience and insight gained during the construction of a simple Internet Relay Chat (IRC) server used internally for debugging chat clients.

The second section addresses text compression. After presenting an overview of the three basic categories of compression, it explores the compression-related issues of state containment and resynchronization in an intermittent-connection environment. It then presents an algorithm for compressing small text strings that is superior to existing text compression approaches, and an algorithm for compressing Extensible Markup Language (XML) text using *a priori* knowledge of the XML schemas on which the XML text is based. The section concludes by unifying the two algorithms to support highly compressed XML-based chat messages.

2. CHAT SERVER ARCHITECTURE

2.1 CHAT SERVER ANATOMY

2.1.1 Crossbar Switch

A basic component of chat server construction is the crossbar switch object. To illustrate its function, consider a simple application where all the users are connected to the same server. The server maintains a list of current users and a list of channels (chat rooms) where at least one user is present. The crossbar switch implements the joins between the users and the channels. Figure 1 shows an example with four users and four channels. The “U” nodes represent users, the “C” nodes represent channels, and the “J” nodes represent joins of a user to a channel (Figure 1).

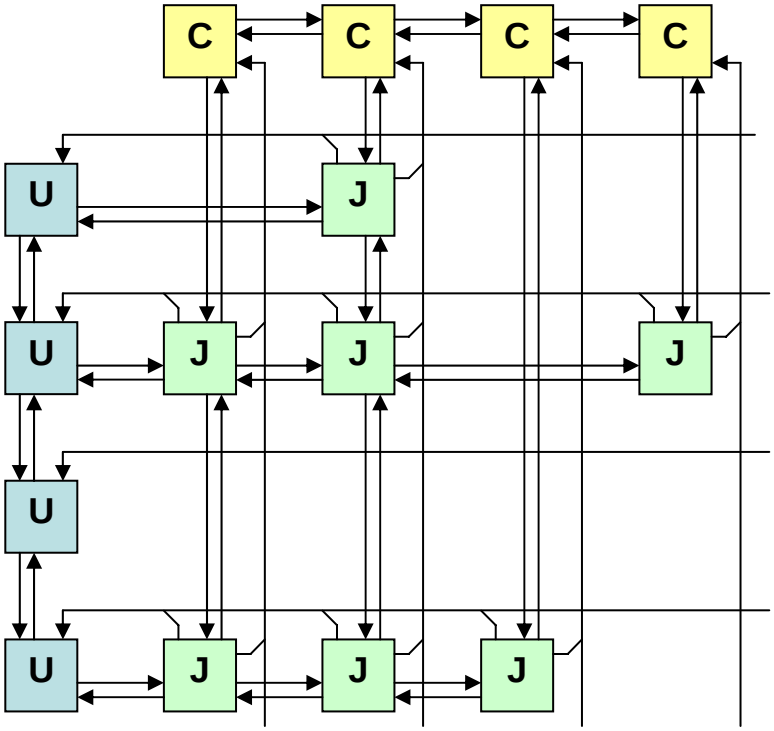

Figure 1. User channel join diagram.

Each join node has a link to the user node and to the channel node it joins. Separate links implement a list of all the join nodes of a given user (shown horizontally) and separate links implement a list of all the join nodes of a given channel (shown vertically). The various linked lists of nodes are implemented using forward and backward links to facilitate the removal of nodes from any position in the respective lists.

A join node cannot exist without a link to a user node and to a channel node. A user node, on the other hand, can have zero or more join nodes attached, and a channel node can have one or more join nodes attached. User nodes, channel nodes, and join nodes have no common attribute types among them. Therefore, the user node, channel node, and join node object classes should be treated as distinct without any hierarchical dependencies of one to another.

Figure 1 gives the impression that the sequential order of nodes in the various linked lists forms a neat grid with respect to one another. This is an oversimplification to make the diagram less confusing. In practice, users can join and part channels randomly with respect to one another over time that can shuffle the sequential order of nodes on the linked lists. The algorithms described below, however, are not affected by this reordering.

When a new user node is to be allocated, the allocation algorithm must ensure that the new user node will not be a duplicate. One could sequentially follow the links from one existing user node to the next, as shown in the diagram, to check the requested user name against each user node visited. If the number of users is quite large, however, this process is inefficient. A better approach for locating a user node given a user name is hashing. First the user name is converted into a hash value. The hash value is then used as an index into a hash table. The hash-table entries specify independent linked lists of user nodes for each respective hash value. Typically, the search time is significantly faster, despite the added time to compute the hash value, since the independent linked lists of user nodes are shorter than a single linked list of all user nodes. A similar approach may be used during the allocation of new channel nodes.

When a new join node is allocated, the allocation algorithm must ensure that another join node does not already link to the same user and channel. In this case, a sequential search of the linked list of all the join nodes of the given user is sufficient because a user will not typically be joined to a large number of channels simultaneously.

Regardless of the search methods employed, the comparisons may need to be case insensitive. IRC, for example, requires all nicknames and channel names to be case insensitive. Case-insensitive hash searches must map all uppercase and lowercase variations of a given name to the same hash value.

User nodes and channel nodes have attributes related to their respective roles. User node attributes include login state, nickname, communication port, visibility to other users, and access privileges. Channel-node attributes include channel name, topic, visibility to users, and access restrictions.

There are two types of join nodes. The first type represents a user who has personally joined the channel. The user receives messages that other users send to that channel and can send messages to other users listening to that channel. The second type of join node represents an invitation by another user for the invited user to join the channel. The join node appears on the linked list of the invited user, not that of the user issuing the invitation. No communication occurs between the channel and the invited user until the invited user accepts the invitation by personally joining the channel. The two types of join nodes are distinguished by an invitation flag attribute resident on the join node.

2.1.1.1 Message Operations

When a user logs into the server, a user node is created and added to the linked list of user nodes (shown at left in Figure 1).

When a user personally joins a channel without an invitation, first the linked list of join nodes for the user (shown horizontally) is checked. If an invitation join node already exists for the requested channel, all that needs to be done is to clear the join-node invitation flag attribute to accept the invitation. If a join node exists for the requested channel that is not an invitation, the user already has joined the channel, so no action is required except perhaps to send a warning message to the user. If no join nodes exist on the linked list, or none for the requested channel, the list of existing channels (shown on top) is searched.

When a user invites another user to join a channel, first the linked list of join nodes for the invited user (shown horizontally) is checked. If an invitation join node already exists for the requested

channel, no action is required. If a join node exists for the requested channel that is not an invitation, the invited user has already joined the channel, so no action is required except perhaps to send a warning message to the user issuing the invitation.

In both cases, if the requested channel does not yet exist, a channel node is created and added to the linked list of channel nodes. A join node is also created. Links are established to connect the user node to the new channel node through the new join node. If, on the other hand, the requested channel already exists, only the join node is created and the respective links established. Finally, the invitation flag attribute of the new join node is initialized to indicate whether it is an invitation or not.

Regardless of the number of repeated invitation or personal join requests of a given user to a given channel, only one join node associates the user to the channel. The user node and channel node must exist, or be allocated, before the join node associating them can be allocated. This requirement can be enforced by requiring the user and channel nodes to serve as parameters to the join-node allocation algorithm.

When the user sends a message to a channel, the linked list of join nodes for that user (shown horizontally) is searched to find the one linked to the desired channel node. Invitation join nodes are ignored. Then the linked list of join nodes for that channel (shown vertically) is sequenced to send a copy of the message to each user identified by the respective join node. Again, invitation join nodes are ignored. The join node for the user who generated the message is also ignored since it does not need a copy of its own message.

When the user “parts” (leaves) a channel, the linked list of join nodes for that user (shown horizontally) is searched to find the one linked to the desired channel node. The join node is removed from its two linked lists (shown horizontally and vertically) and discarded. Then the linked list of join nodes for the desired channel is checked to determine whether any other join nodes for users are still joined or invited to the channel. If none are detected, the channel node also is removed from the linked list of channel nodes (shown at top) and discarded.

When the user logs off or is forced off the server, the linked list of join nodes for that user (shown horizontally) is sequenced to identify all channels to which the user is joined or invited. A “part” (departure) is performed on each respective channel as described above. Finally, the user node itself is removed from the linked list of user nodes (shown at left) and discarded.

2.1.1.2 Presence Operations

Chat user presence is information indicating the extent to which users are participating in the chat environment.

A local user may receive presence information about a target user in two ways. The first way is for the local user to send the server a message explicitly requesting that the server report whether the target user is present. Local users typically do this when they first log into the server or join a channel so they can initialize a display of the current state. The other way is for the server to notify the local user when the server recognized that the presence of a target user has changed, which enables the local user to update the display when necessary.

IRC defines commands to request the server to respond with user presence information. The server compiles the information by traversing the appropriate crossbar switch linked lists, reading the desired information from the visited nodes, formatting the information into messages, and sending these messages back to the requester. Each user node has a bit indicating whether the user is globally visible or invisible to all other users. Invisible user nodes are ignored during the link list traversal.

The server also generates messages indicating when a visible user joins or parts a channel. These messages are sent to all users who have joined the channel so that they can display an undated list of users currently joined on the channel.

User presence in IRC is closely associated with chat room channels. A user can query for another user's presence, but no mechanism enables the server to notify a user of changes in another user's presence unless both users are joined to the same channel.

The Extensible Message and Presence Protocol (XMPP), on the other hand, introduced the concept of a roster. The XMPP user obtains notification of changes in the user presence of the other users listed in its roster.

A list of users in a roster is similar to a list of chat rooms a user has joined. Each can be conceived as a list of channels. Unlike chat room channels, however, the presence channels do not relay user-specific messages. They only supply the user presence information of the respective user to which they are uniquely dedicated. When a user wishes to receive messages from a chat room, the user "joins" the chat room channel. When a user adds another user to its roster, the user "subscribes" to the other user's presence channel, which suggests similar implementations using a crossbar switch object.

The user whose presence the channel node supports is called its owner. Respective join nodes connect the presence channel node to the owner's user node and to the user nodes of all the currently online users who have subscribed to the owner's presence. This connection enables the owner, or the server on the owner's behalf, to send a message indicating a change in the owner's presence to all the other currently online users subscribed to that user's presence.

The presence of a user must be communicated at login time to all the other users who have subscribed to the user's presence within their respective rosters. A user node will exist already for the user logging in when at least one other currently online user is subscribed to the user's presence. Thus, the user logging in must search the linked list of user nodes to ensure that one does not already exist for itself before allocating a new one. This search must be made to ensure that the identity of the user logging in is unique.

The message crossbar switch and the presence crossbar switch share the same linked list of user nodes (Figure 2). This linked list can be implemented by providing the user node with links to its message-join-node linked list and its presence join-node linked list.

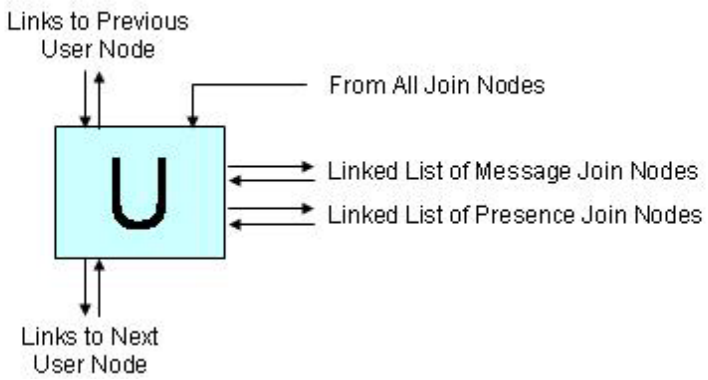

Figure 2. Links to user node.

2.1.1.3 Primitive Methods

In addition to its constructor and destructor, the user node object includes the following methods:

UserNode **getNextUserNode** ( )
UserNode **getPreviousUserNode** ( )

JoinNode **getFirstMessageJoinNode** ( )
JoinNode **getFirstPresenceJoinNode** ( )

UserState **getUserState** ( )
void **setUserState** (UserState)

In addition to its constructor and destructor, the channel node object includes the following methods:

ChannelNode **getNextChannelNode** ( )
ChannelNode **getPreviousChannelNode** ( )

JoinNode **getFirstJoinNode** ( )

ChannelState **getChannelState** ( )
void **setChannelState** (ChannelState)

In addition to its constructor and destructor, the join node object includes the following methods:

JoinNode **getNextUserJoinNode** ( )
JoinNode **getPreviousUserJoinNode** ( )

JoinNode **getNextChannelJoinNode** ( )
JoinNode **getPreviousChannelJoinNode** ( )

UserNode **getUserNode** ( )
ChannelNode **getChannelNode** ( )

JoinState **getJoinState** ( )
void **setJoinState** (JoinState)

In addition to its constructor and destructor, the crossbar switch object includes the following methods:

void **addUserNode** (UserNode)
UserNode **findUserNode** (UserName)
UserNode **getFirstUserNode** ( )
void **removeUserNode** (UserNode)

void **addMessageChannelNode** (ChannelNode)
void **addPresenceChannelNode** (ChannelNode)
ChannelNode **findMessageChannelNode** (ChannelName)
ChannelNode **findPresenceChannelNode** (UserName)
ChannelNode **getFirstMessageChannelNode** ( )
ChannelNode **getFirstPresenceChannelNode** ( )
void **removeChannelNode** (ChannelNode)

void **addMessageJoinNode** (UserNode, ChannelNode)
JoinNode **addPresenceJoinNode** (UserNode, ChannelNode)

JoinNode **findMessageJoinNode** (UserNode, ChannelName)
JoinNode **findPresenceJoinNode** (UserName)
void **removeJoinNode** (JoinNode)

int **getUserNodeCount** ( )
int **getMessageChannelNodeCount** ( )
int **getPresenceChannelNodeCount** ( )

2.1.2 Message Management

2.1.2.1 Message Segments

A chat message is composed of component segments that are distinguished by how they interact with the server.

When a client user is connected directly to the server through a dedicated server port, the messages that the client user sends to the server have one or more destination segments to indicate the intended message recipients, and a payload segment intended for the message recipients to interpret. The message recipients may include other client users, the server itself, or other servers. When the client user has logged into the server, the client user is not required to include its own identity in the messages it sends to the server. The server already knows the client user's identity because it assigned the client user exclusive use of a particular server port.

When the message recipients include other client users or other servers, the recipients may need to know the client user's identity to return a response or error message. The server provides the client user's identity by adding a source segment to the message it receives based on the port from which it came.

When a message recipient is assigned exclusive use of a particular port, the server does not need to include the recipient's identity in the messages sent to the recipient. The recipient already knows its own identity and it would not see messages intended for any other recipient. Thus, the server can strip the destination segment from messages it sends to a recipient assigned to a dedicated port.

When a message passes from server to server on its way to its final recipient, the source segment and the destination segment must be preserved.

The same source-segment content is associated with all the different payload segments that the same client user sends to the server. The same payload-segment content is associated with all the different destination segments of a given message. Thus, a functional independence exists among the segment types. Furthermore, all three segment types can vary in length, which means that usually some copying of segment content would be required if the segments of a message were to be concatenated into a single continuous buffer. The three segment types also differ in longevity. The server generates the source segment content and keeps it as long as the client user remains logged into the server. The payload segment content exists as long as at least one output queue holds or is currently transmitting it. The destination segment content is no longer needed when the message enters the respective output queue of a dedicated port. For all these reasons, the source, payload, and destination segments of a message should be maintained in separate message buffers, as illustrated in Figure 3.

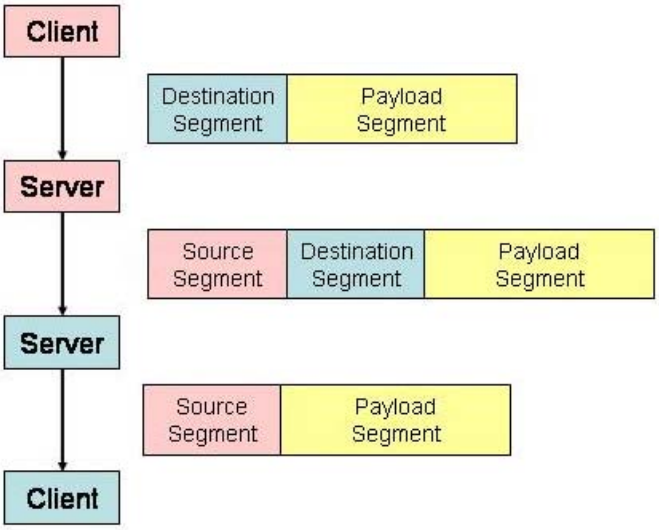

Figure 3. Segments of messages held in separate buffers during transactions.

A message sent from a source user to a local-destination user may specify a channel from among a set of possible channels. Whereas the destination user knows its own user-specified identity, typically, the identity of the channel that the source user used is unknown until it has received the message from the source user. The server is not required to send the destination user its own user identity, but it must send it the specified channel identity. Therefore, the user identity is part of the destination segment, which can be discarded, whereas the channel identity is part of the payload segment, which must be retained.

In the case of a message sent to a destination user located on a remote server, the local server routes the message to a local-destination user it has dedicated exclusively for the connection to the remote server. Since the remote server typically supports many users of its own, the remote destination user identity must be retained in the message that the local server sends to the remote server. Therefore, the local server internally treats the remote-destination user identity as part of the payload segment.

Unfortunately, existing chat-message protocols are not always organized into contiguous segments as described above. Consider, for example, the following source client-to-server XMPP message:

```
<message to=’room@service/nick’ type=chat>
   <body>Hello world!</body>
</message>
```

The “nick” identifier, embedded within the Jabber Identifier, is destination segment information. All the rest is payload segment information or XML structural overhead. Thus, the required sequential location of the destination segment splits the payload segment into a portion before it and a portion after it. The split payload segment can be implemented with a separate buffer for each portion. The link to the destination segment appears in the message-segment vector between the links to the first and last portions of the payload segment.

Treating the entire source-client message as payload segment information may seem easier. In that case, the destination segment information would be transmitted to the destination client even though the client does not need it. The impact on message length is small because the destination segment

information is relatively short. Such an approach, however, fails to consider the other reason for buffer segments, namely, to avoid duplicating payload segment information for each destination client receiving the same payload content.

Individual messages are defined by the sequence of links to the message buffers containing their segments. When a message is placed in a queue, its set of links must remain in the proper sequence and enqueue as an atomic unit, which can be accomplished by representing a message as a vector of message-segment links such as by using the **struct iovec** supported by various operating systems (Figure 4).

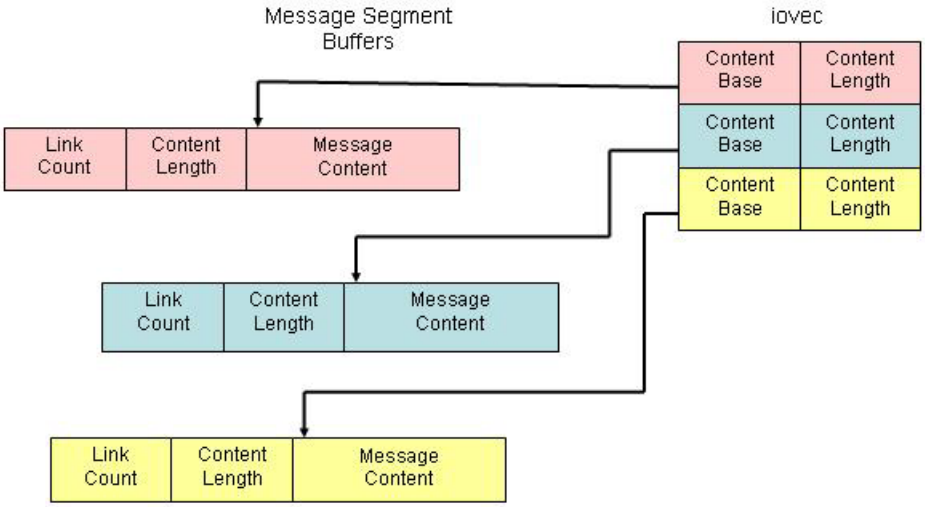

Figure 4. Message as a vector of message-segment links.

2.1.2.2 Buffer Allocation

Each user node has an output queue of links to message buffers. By queuing links to message buffers rather than queuing message content, all users who will receive the same message can share the same buffer. Memory space and execution time are saved since message content does not require duplication.

Each buffer counts the number of user links referencing the buffer. The count is incremented when a buffer link enters a user node output queue and decremented after the buffer content has been transmitted and the link is removed subsequently from the user node output queue. The link counts of any pending buffers also are decremented when the user logs off or is forced off the server. The message content is discarded when the link count reverts to zero.

Although operating system routines can be used to allocate and deallocate memory for individual message buffers, it is more efficient in most cases to push the memory of previously allocated message buffers that are no longer needed to a stack and to pop from that stack before allocating new memory when new buffers are required. Initially, and when the stack is exhausted, memory sufficient for a number of message buffers can be allocated in a single system call and pushed to the stack for current and future use. This same technique can also be used for allocating and deallocating crossbar switch nodes. If the processing environment already implements this functionality, the programmer is not required to duplicate it.

Message segments retained for possible subsequent reuse are called “persistent message segments.” The server’s message-of-the-day, for example, is a persistent message segment. They are kept persistent by ensuring that at least one link to their message buffer exists to keep their message-buffer-link count greater than zero.

Chat room and presence channels may use persistent messages to record channel state or mode changes. The channel typically sends a notification message of each change to the users that are joined to that channel, which enables users to track changes in the channels they have joined as they occur in near-real time. Moreover, a user usually can request the current state or mode of a channel explicitly. If the message content and format in both cases are the same, retaining the notification message beyond the initial distribution makes sense in case a user requests it subsequently.

To help avoid potential race conditions, content of the individual message segment buffer never is altered after it is created. When the state or mode represented by the message content change, a new message segment in a new message buffer must be created. The old message buffer content is discarded only after all links to the old message buffer are released.

2.1.3 Message Distribution

2.1.3.1 User Nodes

The way a message is sent to a user node depends on the user node type. The three types of user nodes are local user nodes, server user nodes, and remote user nodes.

Each local user node supports a directly connected local client, typically a human operator, on a dedicated input/output port. Local user nodes support the client-server chat message protocol.

Each server user node supports one or more remote servers accessed locally through a single dedicated input/output port. Server user nodes support the server–server chat message protocol.

Each remote user node represents a single user on a remote server for which the local server has knowledge. Remote user nodes do not implement an input/output port. Instead, they have a link to the server user node assigned to the server hosting the user they represent.

2.1.3.2 Remote Servers

Several users joined to a channel may reside on one or more remote servers that the local server accesses through a single dedicated input/output port. Sending a duplicate copy of the same channel message to each associated remote user would be inefficient. A better approach is to send a single copy to the channel on the remote servers and let the remote servers distribute the message to each of their own users joined to that channel.

The channel-message-link-distribution algorithm is described below. The channel sequences through its list of join nodes. If the join node is an invitation node, it is skipped as described previously. If the join node is associated with a local user node, the message link is given immediately to the local user node output queue. If the join node is associated with a remote user node, the identity of its server-user node is added, if not already present, to a list in temporary storage. After all join nodes have been visited, the newly constructed list of server-user nodes is sequenced. The message link is given to the output queue of each server user node on the list. Since each server user node appears only once on the list, only a single message is sent to the respective remote server.

The chat protocol may permit the same message to be sent to multiple channels. In that case, the list of server user nodes is not sequenced until the join nodes for all the selected channels have been visited.

2.1.3.3 Multicast Networks

The server applications described thus far have assumed that each local user communicates with the server through an independent input/output port that the server assigns solely to that respective user. Multicasting is an alternative approach where a group of users share the same server port. The users are responsible for listening to the message traffic sent from the server through the port, extracting the messages that apply to themselves and ignoring the rest.

One advantage of multicasting is that when a channel distributes the same message to more than one user in the multicast group, the message only needs to be transmitted once. All the intended users simultaneously recognize the channel that the message identifies as one they are monitoring and simultaneously accept the associated message content. The disadvantages of multicasting are that (1) recognizing intended messages imposes a processing burden on all users, and (2) messages that no user desires may create unnecessary traffic.

From the perspective of the local server, the algorithm to eliminate duplication of the same channel message sent to different users connected through a multicast port is essentially the same as that for different users connected through a port to the same remote server described above. What distinguishes the two cases is the method by which messages are distributed when they leave the local server.

2.2 NETWORK CONNECTIVITY

Human users face two problems when they use chat systems with underlying networks that suffer from intermittent connectivity during a chat session.

The first problem is re-establishing network connectivity. When a chat server detects loss of network connectivity with a client user, it automatically logs out the client user, which prevents a different client user from subsequently connecting to the same port appearing to be the original client user. It also prevents tying up previously reserved resources for a client user who never returns. Thus, when a client user regains network connectivity, it must log back into the server. Ideally, the client–user software should be designed to log back into the server automatically so that the human user does not need to repeat the sequence of steps required for the initial log in.

The second problem is determining if and when messages from a client user reached one or more recipients. The message could be delayed or lost between the client user and its server, between servers, or between servers and potentially any number of recipients. The client-user software can be designed to indicate when sent messages are stuck in the queue waiting transmission to the server. This indication could be displayed to the user by changing the font of the affected message content, such as its color, in the client software window. The server can be designed to generate and return an error message retracing the path back to the original message sender when a roadblock in the path to the recipient is detected.

3. DATA COMPRESSION

3.1 COMPRESSION TECHNIQUES

Data compression research has a history spanning over half a century, has produced scores of published technical papers, and in the past decade, spawned an industry. Three fundamental approaches have emerged: symbol compression, sequence compression, and context compression.

3.1.1 Symbol Compression

Digital communication systems represent individual characters of a character set with unique sequences of bit values. In the ASCII character set, for example, the letters ‘A’ and ‘B’ are represented by the bit sequences ‘1000001’ and ‘1000010’, respectively. Typically, the representations of all the characters in the character set employ the same number of bits. In the ASCII character set, all characters are uniquely represented by 7 bits.

In language text applications such as English, some characters, for example, the space character or the letter ‘e’, are used far more frequently than others, such as ‘X’. Symbol compression translates the character representations into new representations that use fewer bits for characters expected to occur more frequently, and consequently, more bits to cover characters expected to occur less frequently.

Symbol compression is not limited to representations of text characters. It can be generalized to include any sequence of information elements, called symbols, and hence the designation, symbol compression. It can also take non-binary forms such as Morse code, which uses dots, dashes, and gaps.

Huffman coding, based on the work of Shannon’s information theory, is a classic example of symbol compression. The number of bits in a symbol representation can vary widely. The bits forming the symbol representation are assigned such that they can only be interpreted in one way. For example, the letter ‘E’ might be coded by ‘011’. No other symbol will start with the same sequence of bits. The letter ‘A’, for example, might be coded as ‘1110’.

A practical problem arises when a bit is transmitted in error. In the example above, the ‘E’ is coded in 3 bits, namely ‘011’, and the ‘A’ is coded in 4 bits, namely, ‘1110’. If the first bit is transmitted in error, not only is the character corrupted, but the number of bits in its representation, 3 or 4, is corrupted. Consequently, the location of the first bit for the immediately following character, and all subsequent characters, is also corrupted.

Another problem with Huffman coding, as well as with other implementations of symbol compression, is the dependence on *a priori* knowledge of the expected relative frequencies of symbols. If the input symbols all had the same number of bits and were distributed uniformly, the input would expand rather than compress.

UTF-8, commonly used for Internet text transmissions, employs symbol representations that vary in the number of 8-bit bytes rather than the number of individual bits. UTF-8 assumes that the vast majority of input symbols have 7-bit ASCII character equivalents. If the most significant bit of a representation byte is zero, the coded symbol is completely defined by the seven remaining least significant bits of the byte. Otherwise, multiple bytes define the symbol (Figure 5).

UTF-8 for characters with numeric values from 0 through 7F hex

0	B6	B5	B4	B3	B2	B1	B0

UTF-8 for characters with numeric values from 80 hex through 7FF hex

1	**1**	**0**	B10	B9	B8	B7	B6
1	**0**	B5	B4	B3	B2	B1	B0

UTF-8 for characters with numeric value from 800 hex through FFFF hex

1	**1**	**1**	**0**	B15	B14	B13	B12
1	**0**	B11	B10	B9	B8	B7	B6
1	**0**	B5	B4	B3	B2	B1	B0

Figure 5. Example of the use of “1” and “0” as most significant bit in UTF-8.

WBXML provides as example of non-text symbol compression that represents its binary integers by a sequence of bytes. The most significant bit of each byte, called the continuation bit, indicates whether the byte that immediately follows it is part of the same integer. Its compression is based on the assumption that most input integers have small positive values and hence require few bytes (Figure 6).

WBXML multi-byte integer for values from 0 through 7F hex

0	B6	B5	B4	B3	B2	B1	B0

WBXML multi-byte integer for values from 80 hex through 3FFF hex

1	B13	B12	B11	B10	B9	B8	B7
0	B6	B5	B4	B3	B2	B1	B0

WBXML multi-byte integer for values from 4000 hex through FFFFF hex

1	B19	B18	B17	B10	B16	B15	B14
1	B13	B12	B11	B10	B9	B8	B7
0	B6	B5	B4	B3	B2	B1	B0

Figure 6. Example of the use of “1” and “0” as most significant bit in WBXML.

3.1.2 Sequence Compression

Sequence compression seeks to replace sequences of symbols, rather than individual symbols, with representations requiring less space. Authors essentially practice this compression when they use commonly understood acronyms and abbreviations such as “USA” for “United States of America.”

Ziv-Lempel coding, the grandfather of most modern general-purpose file compression programs, watches for input symbol sequences that have previously occurred, and when found, replaces the repeated sequence with a reference to its previous location and length. Compression results when the reference takes less space in the output compared to repeating the symbol sequence itself. Unlike symbol compression, this approach does not require *a priori* knowledge of relative frequencies specific to an application. It is suitable for all kinds of data formats.

The algorithm has some limitations. Short input sequences may compress poorly, or not at all, since they typically are not long enough to exhibit many repeated internal sequences. Long input sequences may compress very slowly because of the time required to search for previously encountered internal sequences. Subsequent research has focused on the searching problem.

A significant breakthrough was the discovery that the letters forming individual words extracted from a text file could be stored and accessed efficiently using a tree structure (Figure 7). Each node of the tree, except for the root, specifies a letter. By traversing from a leaf node to the root one visits the associated letters spelling out the word in reverse order. The compressed output indicates the sequence of desired words by listing the references to their respective leaf nodes in the tree. During decompression, a first-in, last-out stack restores the letters of each word to normal order.

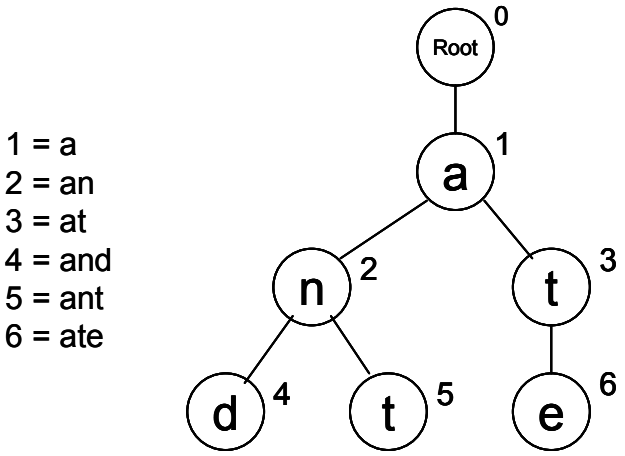

Figure 7. Tree structure for storing words in a text file.

The tree can be constructed as the input sequence is read. Since the tree content consequently depends on the input sequence, it must be included with the compressed output for the compressed output to be understood. Alternatively, when most of the vocabulary that the input uses is known *a priori*, such as words in the English language, a standard tree can be employed. Since the compression algorithm and the decompression algorithm have access to copies of the same standard tree, including it with the compressed output is unnecessary. Furthermore, during decompression, the tree leaf node references can be mapped directly into normal letter order spellings using a lookup table derived from traversals of the standard tree.

Sequence compressed output typically contains a mixture of references to previously defined symbol sequences and non-compressed sequences called literals. Literals are appropriate when they require less space than a reference to them. Literals are required when a standard tree does not support them. The compressed output encoding must support a mechanism to differentiate between inline occurrences of symbol references versus literals.

Approximately one out of every five characters in English text is a space character. Word-oriented languages such as English have established rules for placing space characters between words, and in relationship to other punctuation marks. The space characters do not need to be represented in a sequence of leaf node references representing words since these rules can be used to reinsert the space characters during decompression. Only the space characters between words in a sequence of words represented as a literal must be encoded.

Sequence compression can be combined with the symbol compression. For example, the standard tree sequence compression approach can be used to generate a sequence of tree-leaf node references for English words. These leaf references can then be translated into coded symbols of various lengths, depending on the expected relative frequency of each respective English word, using symbol compression.

3.1.3 Context Compression

Context compression takes advantage of situations where the input syntax require that only certain symbols or symbol sequences follow given symbols or symbol sequences.

The typewriter keyboard provides an analogy. The meaning of the character keys depends on whether the shift lock is on or off. Pressing the shift-lock key defines the context as uppercase or lowercase. By including the shift-lock key, the number of keys required to implement the typewriter keyboard is reduced nearly in half. The clef symbols in music notation perform a similar space-compressing function. The shift register key on the clarinet that the left thumb actuates provides a dynamic analog to the typewriter keyboard example.

An important application of context compression is XML. The syntax rules are defined by the XML specification and by the applicable XML schema. For example, when the schema defines the set of possible valid attribute and child element names that can reside within the context of a given parent element name, one only needs to define unique coded symbols to distinguish among those attribute names, child element names, the end-or-element tag, and possibly an escape for a syntax error. Numeric and string-type attribute names themselves define a context for the attribute values they specify. The combination of an enumeration-type attribute and its value, on the other hand, may be treated as a unit and assigned to a unique coded symbol value.

3.2 INTERMITTENT CONNECTIONS

Communication environments suffering from frequent disconnects and reconnects during a chat session present special challenges to the selection of a compression algorithm.

Most compression algorithms, in one way or another, incorporate previously established state information when encoding current input. If connectivity is lost during transmission of a portion of the encoded stream that defines state information, everything depending on that state information following the lost portion is affected. Huffman coding provides one of the worst cases of the problem. As shown in a pervious section, a single bit error can corrupt everything following it. Loss of a single bit has the same impact as an error bit when its state information is guessed incorrectly or it is assumed not to exist.

State information in a data stream is characterized by its presence, its location, and its value within the data stream. Mitigation of intermittent communications requires knowing whether state information has been lost, knowing where to look for what was lost, and knowing what was lost.

Loss of state location can occur when the interpretation of symbols depends on their relative distance from other symbols. Consider, for example, a compression algorithm that replaces repeated subsequences with links defined by the distance back to the nearest uncompressed occurrence of the respective subsequence. If a portion of the data stream of unknown length is lost between the uncompressed subsequence and the subsequent link referring to it, the distance defined by the link would point to the wrong location. Although the link still physically exists, the location of the state it represents, namely the uncompressed subsequence, has been lost. In general, compression algorithms encoding relative distances should be avoided in intermittent communication environments.

3.2.1 State Containment

Intermittent communication connections are characterized by large gaps of adjacent symbols in the data stream. If a particular symbol is lost, the symbols adjacent to it are likely to be lost. If a particular symbol is present, the symbols adjacent to it are likely to be present. Thus, a symbol referencing state information that is adjacent or close is more reliable than a symbol referencing state information that is separated by some distance.

State containment seeks to improve reliability by allowing a symbol to reference state information only from other symbols located within the bounds of a defined range. Recovery is simplified because states outside the defined bounds do not need to be retained and states within the defined bounds are in close proximity.

The primary disadvantage of state containment is that limiting the states that can be referenced also limits the opportunities to compress using available states. When the defined bounds of a single chat message define the range, for example, few if any repeated subsequences provide states for compression.

3.2.2 State Synchronization

Compression algorithms where the meaning of a symbol depends on the state of one or more previous symbols imply the existence of an underlying state machine. When input data stream reception is restored after the possible loss of symbols, the current state of the state machine must be re-established. There is no problem if all the lost symbols can be recovered since the state machine can sequence through them. But if this is not possible or convenient, or if symbols may be in error, some other mechanism is needed to synchronize the state machine with the input data stream.

Synchronization can be re-established by including symbols, called synchronization symbols, within the input data stream. Synchronization symbols have both unique codes and simultaneously do not depend on the state of any previous symbols. When a synchronization symbol is encountered, the state machine goes immediately to the state associated with that synchronization symbol regardless of what preceded. Everything that follows is synchronized as long as it does not reference anything prior to the synchronization symbol. State containment should use a synchronization symbol at the beginning of the containment range. They can, of course, be used elsewhere as well.

Synchronization symbols have other uses besides synchronization. They can have additional meanings as long as they do not depend on other symbols. But because their codes must be unique, the number of codes assigned to synchronization symbols should be restricted to things that are likely to occur frequently.

3.2.3 State Codebook

State containment, described above, applies to compression that uses links in the communications data stream to prior state information in the same communications data stream. The codebook approach, on the other hand, uses links to state information maintained in an independent table called the codebook. The sender and the receiver have a copy of the codebook. Only the links need to be sent within the communications data stream because both the sender and the receiver already know what each link represents from their respective local copy of the codebook.

An obvious limitation of the codebook approach is that it can only compress what is already predefined in the codebook. Chat text, unlike literary text, tends to be spontaneous. Sloppy spelling and neologisms are unsupported. Since it is unlikely that the codebook will cover everything, the

codebook approach must provide a means of inserting segments of uncompressed text, called literals, or perhaps text compressed by some other means.

The codebook approach is particularly effective in compressing XML element and attribute tags, and attribute enumeration values such as encountered in the XMPP. XML namespace definition attributes typically contain a large number of characters. Replacing a namespace attribute tag and value with a single codebook symbol can dramatically reduce the size of an XML message.

3.3 SMALL TEXT COMPRESSION

The previous section described the three fundamental techniques that can be used, individually or in combination, to compress data. This section applies these techniques to create a new compression algorithm called Small Text Compression.

Various sequence compression algorithms are available for compressing the size of data files. These algorithms compress by searching for repeated subsequences within the input sequence and reorganizing the input sequence so that these subsequences are recorded only once. They work best when the input sequence is large and contains repeated subsequences. Their performance on short sequences such as text messages encountered in chat applications is poor. As is explained below, they usually expand rather than compress the size of short text sequence. Small Text Compression is tailored to compress short text sequences, filling the gap left by traditional file compression algorithms.

3.3.1 Compression Design

The Small Text Compression algorithm exploits the observation that most of the characters in a text message come from a limited set of alphabetic characters. Characters expected frequently are coded with fewer bits than those expected rarely. The algorithm also exploits the observation that uppercase and lowercase characters are rarely randomly intermixed. If a given character is lowercase, for example, it is unlikely that the immediately following character in the sequence will be uppercase or a decimal digit.

The set of possible input text characters is partitioned into character subsets. Uppercase characters are in one subset, lowercase characters are in another, and decimal digits, punctuation marks, and symbols are in a third. A fourth character set can handle characters rarely expected. Special non-printable characters, called state characters, are inserted into the compressed text sequence to indicate transitions from one subset to another. The space, end-of-line, and end-of-message characters are available in all subsets. Compression is normally achieved because fewer bit positions are needed to represent characters from a subset than to represent character from the full set, and also because more bit positions are saved than lost by the insertion of occasional state characters.

A typewriter-style keyboard uses a similar approach. As explained in Section 3.1.3, the keyboard does not have separate keys for uppercase and lowercase versions of the same letter. The shift key determines the case selected. When the algorithm inserts a state character, it is like the typist setting or clearing the keyboard shift lock. The shift lock affects neither the space bar or carriage return. Similarly, the state character affects neither the compressed space character or end-of-line characters.

The uppercase and lowercase character subsets must support 26 letters of the alphabet, the space character, the end-of-line character, and the end-of-message character, for a total of 29 possible characters. These characters can be represented uniquely with 5 bits. The end-of-message character is included to mark the end of a sequence of 5-bit compressed characters because often the end of the sequence does not align with the string of 8-bit bytes used to store it. Table 2 lists the compressed

character code assignments. Each column specifies a subset. Binary numbers between "<" and ">" symbols represent state characters.

Table 2. Compressed 5-bit character codes.

Subset Character		27	28	29	30
		<11011>	<11100>	<11101>	<11110>
0	00000	End-of-Message			
1	00001	**A**	**a**	**1**	**!**
2	00010	**B**	**b**	**2**	#
3	00011	**C**	**c**	**3**	**$**
4	00100	**D**	**d**	**4**	**%**
5	00101	**E**	**e**	**5**	**&**
6	00110	**F**	**f**	**6**	@
7	00111	**G**	**g**	**7**	
8	01000	**H**	**h**	**8**	
9	01001	**I**	**i**	**9**	
10	01010	**J**	**j**	**0**	
11	01011	**K**	**k**	**+**	
12	01100	**L**	**l**	**-**	
13	01101	**M**	**m**	*****	
14	01110	**N**	**n**	**/**	
15	01111	**O**	**o**	**=**	
16	10000	**P**	**p**	**<**	000xxxxx
17	10001	**Q**	**q**	**>**	001xxxxx
18	10010	**R**	**r**	**.**	010xxxxx
19	10011	**S**	**s**	**,**	011xxxxx
20	10100	**T**	**t**	**;**	100xxxxx
21	10101	**U**	**u**	**:**	101xxxxx
22	10110	**V**	**v**	**?**	110xxxxx
23	10111	**W**	**w**	**(**	111xxxxx
24	11000	**X**	**x**	**)**	
25	11001	**Y**	**y**	**'**	
26	11010	**Z**	**z**	**"**	
27	11011	space	<11011>	<11011>	<11011>
28	11100	<11100>	space	<11100>	<11100>
29	11101	<11101>	<11101>	space	<11101>
30	11110	<11110>	<11110>	<11110>	space
31	11111	End-of-Line			

Since 5 bits provides 32 possible codes that are listed in Table 2, the remaining three codes are available to represent state characters. A state character does not need to be inserted into a compressed sequence that is already in the desired subset. Only state characters that change the subset need to be included in the current subset. Thus, all four subsets are supported, namely, the current subset, and transition to any of three other subsets using the three remaining characters of the current subset as state characters.

The first compressed character of the sequence must be a state character introducing one of the four respective subsets, an end-of-line character, or an end-of-message character. The end-of-message character indicates a null message. The algorithm could have been designed so that an initial subset is assumed by default, which would make the initial state character unnecessary when the default subset is applicable. The alternative of requiring that the subset be coded explicitly was

selected so that the remaining 26 possible first 5-bit compressed character codes would be available for future expansion.

The Small Text Compression algorithm supports all the 256 possible 8-bit input characters. Table 3 shows an example. The various subsets provide 5-bit codes for the most likely encountered 8-bit input characters. All 8-bit input characters can be coded as a sequence of two 5-bit compressed characters. The first 5-bit compressed character specifies the three most significant bits of the 8-bit input character. The second 5-bit compressed character is identical to the five least significant bits of the 8-bit input character. The sequence of two 5-bit compressed characters is called a literal. Although every 8-bit input character can be represented as a literal, it makes no sense to use a literal when a subset includes a single 5-bit character code for the same 8-bit input character.

Table 3. Compression example.

27	8	28	5	12	12	15	28	23	15	18	12	4	30	1	0
UC	**H**	*LC*	**e**	**l**	**l**	**o**		**w**	**o**	**r**	**l**	**d**	*SC*	**!**	*EM*

UC = Uppercase, *LC* = Lowercase, *SC* = Symbol, *EM* = End-Of-Message

3.3.2 Compression Implementation

The compression algorithm requires that the software implementation pack 5-bit compressed characters into 8-bit bytes. Since “8” is not divisible by “5,” a 5-bit compressed character could be aligned on any of eight possible bit positions within the current 8-bit byte, and in five cases, spill over into the next sequential 8-bit byte. Each of the eight cases involves different instructions, which may include shifting, masking, and outputting a completed byte. This compression can all be implemented by creating a software state machine with eight states, one for each respective bit position alignment case. A “switch” statement can be used to select the instructions needed for each respective case and to specify the next case in the sequence. Provision must also be made for flushing the last 8-bit byte when the 5-bit compressed character sequence does not end on an 8-bit byte boundary.

Converting a compressed sequence back into its original uncompressed form requires the software implementation to unpack 5-bit compressed characters from 8-bit bytes. Another software state machine with eight states, one state for each respective bit alignment case, can also be used here. Provision may also be needed for an unexpected end-of-file or other terminator exception prior to receiving the final end-of-message 5-bit compressed character.

3.3.3 Synchronization Design

Some applications may need to tap into a stream of transmitted messages at an arbitrary point, which requires the message receiver to locate the bit alignment of 5-bit characters in the received bit steam and identify the current character subset. It is assumed that the application is interested only in complete messages and that any received bits prior to the beginning of the first complete message can be ignored.

The end-of-message character is represented by five “`0`” bits. But if the 5-bit character alignment is unknown, one cannot simply search the message stream bit-by-bit for the first sequence of five adjacent “`0`” bits. Many combinations of adjacent 5-bit characters have the concatenation of the least significant bits of the first character followed by the most significant bits of the next character encompass a sequence of five or more adjacent “`0`” bits. Encoding the sequence “PA”, for example, produces the sequence, “`10000 00001`.” It contains eight adjacent “`0`” bits. The eight adjacent “`0`” bits encompass four possible sequences of five adjacent “`0`” bits. None of these four, however,

were intended to represent an end-of-message 5-bit character, even though they each consist of five adjacent “0” bits.

The end-of-message character is the only 5-bit character with five “0” bits. All other 5-bit characters have four or less “0” bits. Thus, the only way that nine adjacent “0” bits could appear in sequence is when an end-of-message character exists somewhere within the sequence of nine “0” bits. This sequence gets one closer to the goal of finding the end of the current message, but still leaves the bit alignment of 5-bit characters unresolved.

Enforcing two rules can solve the problem. The first rule is that all messages supporting synchronization must end with a sequence of at least nine adjacent “0” bits to avoid confusion with adjacent 5-bit sequences that form eight adjacent “0” bits like the “PA” example above. For convenience, implementations may pad the sequence with “0” bits to the next byte boundary, although this is not a requirement as long as the stream contains at least nine adjacent “0” bits. The second rule is that the next message must begin with a state character. The most significant bit of all state characters is a “1” bit. The receiver simultaneously synchronizes with the 5-bit character alignment and the beginning of a new message by searching for the first “1” bit after a sequence of at least nine adjacent “0” bits.

The second rule could be relaxed slightly to allow a new message to begin with any 5-bit character whose most significant bit is a “1” bit. In that case, messages that begin with a non-state character would need to assume an initial default character subset. The advantage of requiring that the first 5-bit character be a small text compression state character is that it leaves room for other first character encodings to introduce alternative message formats.

3.3.4 Synchronization Implementation

The 5-bit character synchronizer state machine is responsible for locating the bit position boundary between Small Text Compression 5-bit characters within a received message byte stream by locating the first “1” bit after a sequence of nine or more adjacent “0” bits. The bit position of the located “1” bit is then used to initialize the 5-bit character reader state machine described previously.

It is assumed that message byte stream bytes are received most significant bit first. When detected, a sequence of at least nine adjacent “0” bits followed by a “1” bit spans either two, or in one case, three adjacent bytes. The three-byte case occurs when the first received byte has a “0” in its least significant bit position, the next byte has a “0” in all eight bit positions, and the last received byte has a “1” in its most significant bit position.

The synchronization state machine contains a 3-byte shift register to store the currently received byte, the previously received byte, and the one received before the previously received byte. The shift register is clocked each time a new byte is received.

The state machine checks each newly received byte to determine the number of adjacent most significant “0” bits it contains. The check can be performed by a loop that counts “0” bits starting at the most significant bit and proceeding until reaching a “1” bit or running out of bit positions. A faster approach is to use a binary search using bit-position masks or equivalent arithmetic comparisons to constants. The first test checks for at least four adjacent most significant “0” bits, and depending on the result, the next test checks for at least six or at least two adjacent most significant “0” bits, and so on.

If the most recently received byte contains only "0" bits, nothing further can be done since the first "1" bit following the "0" bits has not yet been located. Otherwise, the number of adjacent most significant "0" bits in the current byte indicates what must be tested in the previous bytes to obtain an overall sequence across the byte boundaries of at least nine adjacent "0" bits before the "1" bit. The preceding binary search to determine the number of adjacent most significant "0" bits in the current byte automatically leads to separate locations in the software code for each case. To branch on a switch statement based on the number "0" bits found in the current byte is not necessary.

Before receiving the first byte from the message byte stream, the least significant bit of the current shift register byte is set to "1." This setting prevents the false detection of nine adjacent 0 bits when only the first byte is shifted into the shift register.

When the synchronization sequence of bits has been detected, control is transferred to the 5-bit character reader state machine. The 5-bit character-reader state is initialized based on the bit position of the most significant "1" bit in the current byte.

Both state machines can use the same shift register. Note, however, that the clocking of the shift register differs. The 5-bit character-synchronizer state machine clocks the shift register each time a new byte becomes available from the message byte stream. The 5-bit character-reader state machine clocks the shift register when the application requests a new 5-bit character and that character crosses a byte boundary. The 5-bit character reader state machine also only needs two of the shift register bytes. See Tables 4 and 5.

A Java™ language implementation of the 5-bit character synchronizer and reader algorithms can be extended from the **java.io.FilterInputStream** class. Although this class supports **mark** and **reset** methods, their intended semantics are inappropriate for synchronization. Since not all input applications require synchronization, a new method, called **sync**, should be defined for that purpose.

Table 4. 5-bit character synchronizer state machine.

State	When This Bit Sequence Detected																							
	Previous-Previous Byte								Previous Byte								Current Byte							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
8								0	0	0	0	0	0	0	0	0	1							
7	Not Used								0	0	0	0	0	0	0	0	0	1						
6										0	0	0	0	0	0	0	0	0	1					
5											0	0	0	0	0	0	0	0	0	1				
4												0	0	0	0	0	0	0	0	0	1			
3													0	0	0	0	0	0	0	0	0	1		
2														0	0	0	0	0	0	0	0	0	1	
1															0	0	0	0	0	0	0	0	0	1

State 0 is initial state if no synchronization required.
State 8 is not required if no synchronization required.

Table 5. 5-bit character reader state machine.

Current State	Shift Required	Next State		Construct 5-Bit Character from Bits															
		Got EOF	No EOF	Previous Byte								Current Byte							
				7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
8	No	-	3	Not Used								●	●	●	●	●			
7			2										●	●	●	●	●		
6			1											●	●	●	●	●	
5			0												●	●	●	●	●
4	Yes	Exit	7					●	●	●	●	●							
3			6						●	●	●	●	●						
2			5							●	●	●	●	●					
1			4								●	●	●	●	●				
0			3	Not Used								●	●	●	●	●			

3.3.5 Performance

The algorithm typically replaces the vast majority of 8-bit input text characters with 5-bit compressed characters. Consequently, compression may not exceed a ratio of 8 to 5. In practice, however, the compression is less because of the insertion of state characters and possibly padding to align with an 8-bit byte environment. For English text, the size of the compressed output is typically two-thirds the size of the original text input.

The Small Text Compression algorithm is intended only for short text sequences. The test results above provide a general idea of what constitutes “short” for English text. The Small Text Compression algorithm is vastly superior to the gzip and zlib algorithms for a text sequence less than 100 characters. In that range, the gzip algorithm expands rather than compresses. The algorithms are competitive in the 200- to 300-character range. Remember, however, that the results will vary with the nature of the sequence under test. See Figure 8 for a comparison of compression algorithms. Figure 9 graphs the effect of zlib level.

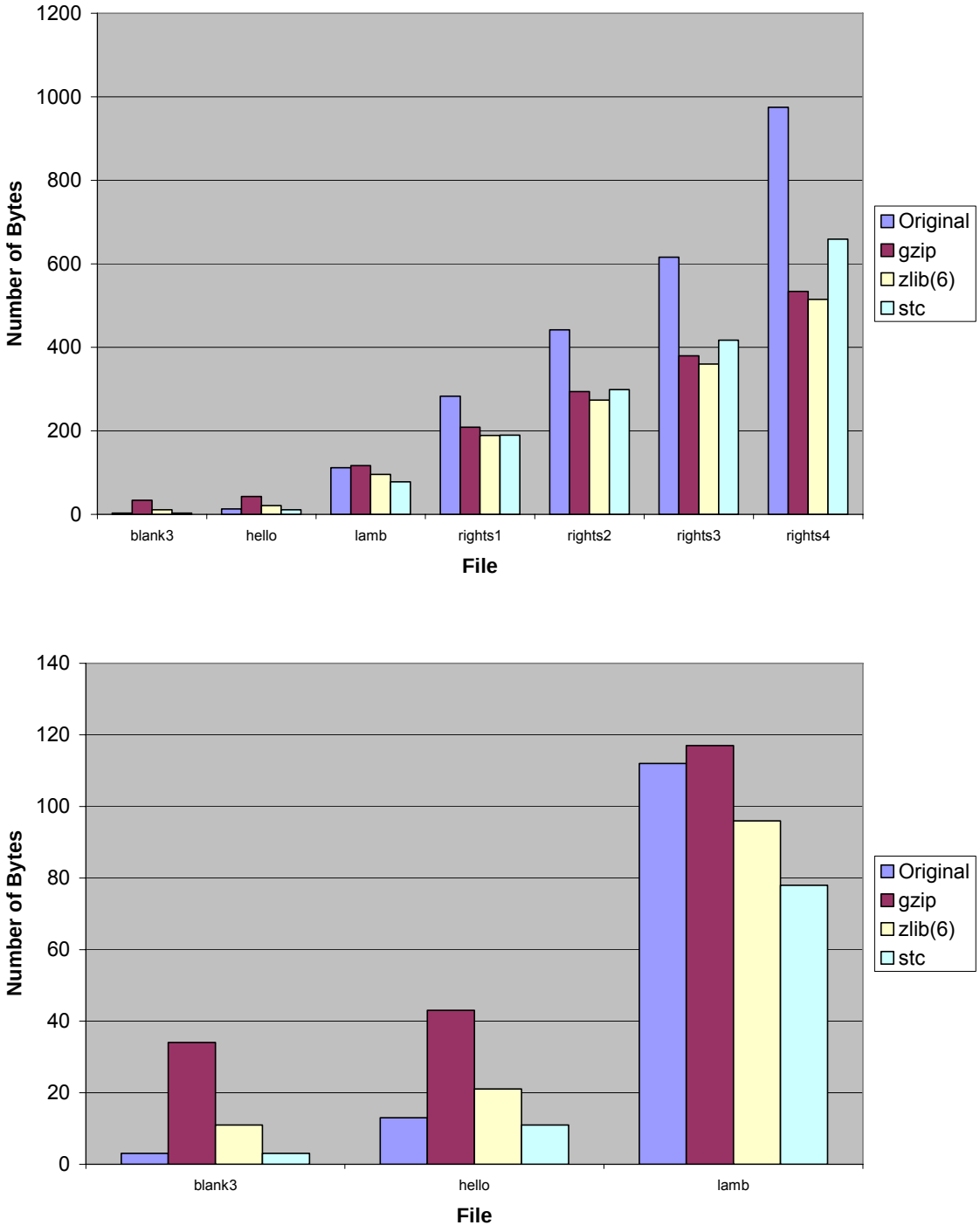

Figure 8. Compression algorithm comparison.

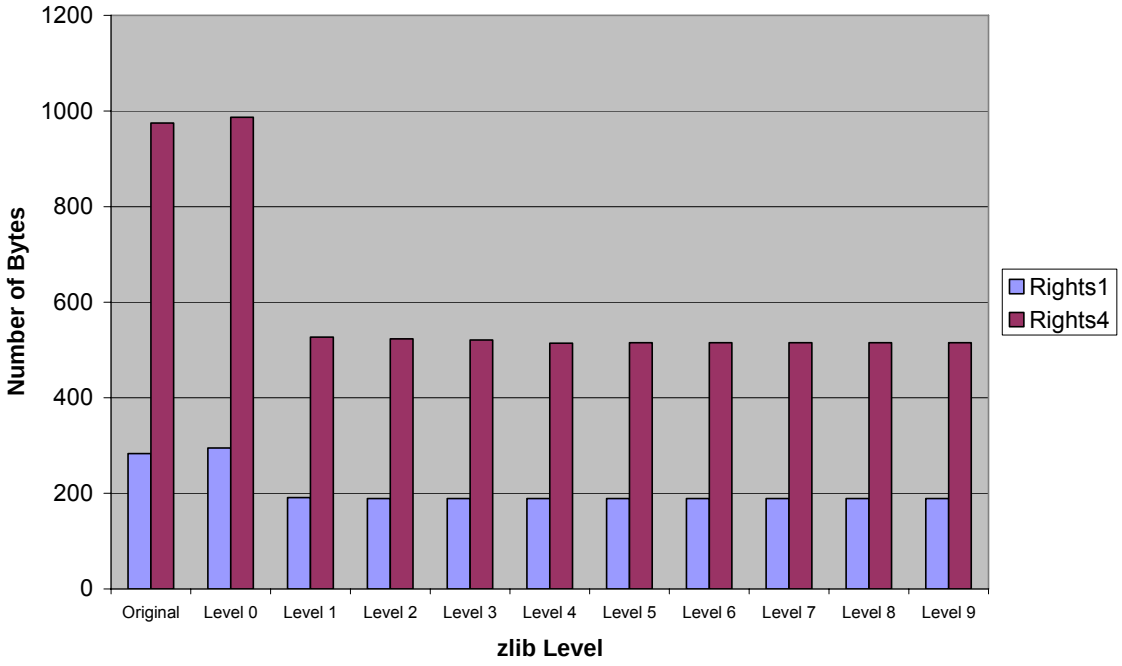

Figure 9. Effect of zlib.

3.3.6 Limitations

Some short text applications may not benefit from compression. In a disk storage application, for example, files usually are assigned exclusive use of fixed-size disk data blocks. No disk space is gained by compressing a short file whose uncompressed size already fits entirely within a single-disk data block. From the perspective of the disk, both the compressed and the uncompressed forms require the same space regardless of the extent to which the disk data block is filled internally.

Compression of short text does benefit network message throughput. Even here, however, some limitations arise in the underlying network protocol layers that cannot be compressed and still remain compatible. The “Hello world!” example above illustrates the problem. It requires 16 5-bit characters for a total of 80 bits. The 80 bits fit exactly into 10 8-bit bytes. Without compression, the same message requires 13 8-bit bytes since it contains 12 text characters and a terminator. Compression has reduced the message size by 3/13 or approximately 23 percent. But when we add the minimum 20 bytes for a TCP envelop and the minimum 20 more bytes for an IPv4 envelop, the results are not so impressive. The uncompressed message is now 53 bytes long and the compressed version is now 50 bytes long. Compression of the data content of the message has reduced the overall TCP/IP message size by 3/53, or approximately 5.7 percent. The percentage improvement is even lower for network bandwidth because of the network hardware requirement for time to separate and synchronize messages from different sources.

These observations may explain why traditional compression algorithm research on compressing short sequences has received little attention. The fact that they are already short tends to imply that compression is unnecessary.

3.4 XML-SCHEMA-BASED COMPRESSION

XML messages tend to be verbose in comparison to equivalent fixed message formats for several reasons. XML messages use a hierarchy of element tags rather than a single identifier to specify message type and function. Element tags appear in fully spelled-out pairs to delimit their scope rather than relying on the relative position of user content in the message. Attributes are specified by name rather than relying on their relative position as well. Finally, Universal Resource Identifiers (URIs), which require a large number of characters, may be embedded rather than assuming predefined formats.

On the other hand, XML has the advantage of extensibility and universality. Elements of new namespaces can be encapsulated within the scope of old namespace elements, which provides a flexible and well-defined way to extend the definable message syntax. XML also provides a universal way to handle message syntax in general. XML promotes software reuse by avoiding unique parser implementations. It encourages the creation and use of proven development tools and object libraries such as editors, syntax event libraries, marshaling interfaces, and database interfaces to serve a wide array of message applications in a uniform manner.

This section presents an XML-compatible method to gain the benefits of the brevity of fixed message formats and the extensibility and universality of XML message formats through XML message compression.

3.4.1 Name Tag Translation

The valid syntax for XML messages can be defined using XML schemas. The schemas, which are written in XML format, specify the target message element and attribute tag names, their hierarchical relationship to one another, the restrictions on their combination and sequence, and the syntactic restrictions on the user content they encapsulate or identify.

In his master's thesis from the Naval Postgraduate School, Ekrem Serin presents a compression method consisting of replacing each element begin, element end, and attribute tag with unique integers. A translation table is generated easily from the schemas by scanning them for <element name=…> and <attribute name=…> schema elements. When the identical schemas and the identical scanning algorithm are available to the transmitter and receiver of the message, they do not need to be included in the translation table as part of the message.

Because XML elements are nested, a stack can represent the nested relationships. A sequential encounter of a child element begin name tag pushes the parent element name unto the stack. A sequential encounter of a child element end name tag pops the parent element name from the stack. The child element end name tag does not need to specify the parent element to which it returns control because the parent's identity is available from the stack.

This process suggests a possible improvement to the assignment of unique integers to tags. The message transmitter encodes the element begin name tags using unique integers as described above. Assuming that the message receiver has a stack, the message transmitter can encode all element end name tags by a single unique integer. Element begin name tags, for example, could be differentiated by unique integers greater than one, and the integer one could represent all element end name tags. The integer assigned to represent all element end name tags, one in this case, is called the "common end tag integer."

Since unique integers no longer are required for each possible element end name tag, the approach reduces the number of required unique integer values for element names by half plus the one extra for the common end tag integer. Element name tags must be distinguished from attribute name tags. Consequently, the inclusion of required additional unique name tag integers for attribute name tags

reduces the overall savings to less than half. The savings are, at best, 1 bit in the name tag integer value, which is hardly worth the trouble. However, as is explained below, the approach can be combined with another approach to reduce significantly the number of bits required.

Attributes have a value and a value type. The number of bits required to transmit the attribute value is determined implicitly from the value type, such as “short” integers that imply 16 bits, or explicitly from the coding of the value itself, such as by a character string delimiter. Thus, the common end tag integer is not needed to terminate attributes.

Element attributes can be viewed hierarchically as children of the element that they describe. Attributes do not, however, have children of their own. Since attributes would never be the parent to something else, their attribute name tag integers would not be stacked.

3.4.2 Contextual Name Tag Translation

Fortunately, more can be done. The schema defines the attributes and elements that are permitted to exist as children of each parent element. Although the total number of attributes and elements that a schema defines may be quite large, the number of attributes and child elements for each parent element individually is generally quite small. Additional compression potential may be obtained by limiting the scope of name tag integer uniqueness to the scope of the respective parent element.

The meaning of an attribute or element name tag integer depends on its parent element. But since the parent element also is represented by a name tag integer, the meaning of the parent element name tag integer depends on its own parent, the grandparent of the original element. The chain continues until it reaches the root element. In general, the meaning of a given name tag integer depends on its entire element ancestry.

The stack maintains the name tag integer element ancestry. The ancestry is defined by all the elements that are on the stack and by their order on the stack. One could determine the meaning of a name tag integer by examining all the elements on the stack in sequence, starting with the element pushed first, which is located on the bottom of the stack. A more practical approach, however, predefines a unique integer for each possible valid element ancestry that the stack can capture so that only a single integer needs examination. By pushing to the stack element ancestry integers rather than name tag integers, only the top of the stack needs examination to determine the entire state of the stack.

The element ancestry integers can be enumerated by traversing all the valid paths through the XML schema, starting with the integer one. The integer zero is reserved for the root element that has no ancestors. Assuming that the enumeration algorithm and the schema remain the same, the enumeration needs to be established only once. The number of bits required to represent element ancestry integers is not an issue because they are never transmitted in the message.

The element ancestry integer on the top of the stack indexes an element ancestry lookup table. Each entry in this table points to a respective name tag lookup table. The name tag lookup table is indexed by the name tag integer under consideration. The name tag lookup table entries define the associated name tag character string, the value type or no value present, and the element ancestry integer to be pushed to the stack if the name tag integer is for a child element. Since character strings may vary significantly in length, the table employs pointers rather than character string literals to save space.

In addition to the field pointing to the respective name tag lookup table, the element ancestry lookup table entries also contain a field pointing to the respective element name character string. As is explained below, the message decoding algorithm needs this field to identify the current element name when decoding common end tag integers. It is also handy for manually interpreting source code listings and when the stack state is displayed for debugging.

3.4.2.1 Coding Algorithm

The message transmitter compresses the XML input name tags as follows.

First it determines from the XML syntax whether the next sequential input name tag is an element begin, attribute, or element end name tag.

When the name tag to be coded is an element begin or attribute name tag, the transmitter searches the current name tag lookup table for the name tag character string that matches the element or attribute name tag. A constant offset is then added to the name tag lookup table entry index where the name tag was found and the sum is transmitted as the name tag integer. The constant offset is necessary to skip over the name tag integers that have a global meaning such as the common end tag integer.

If the name tag was for an attribute, the attribute value would be coded using the value type indicated by the name tag lookup table entry as a guide on how it should be coded. If, on the other hand, the name tag was for a child element, the element ancestry integer from the name tag lookup table entry would be pushed to the stack. The new element ancestry integer now on the top of the stack switches the current name tag lookup table to the one associated with the child element context for use by the name tag integers that follow.

When the name tag to be coded is an element end name tag, the common end tag integer is transmitted and the stack is popped. The stack pop uncovers the previous element ancestry integer, which, in turn, restores the parent name tag lookup table as the current name tag lookup table for use by the name tag integers that follow.

XML syntax permits an element with no child elements to be presented with a single name tag terminated by a slash character. XML parsers usually convert this element into two separate events, one to indicate the beginning of the element and another to indicate the end of the element. The name tag integer coding should follow this same pattern.

3.4.2.2 Decoding Algorithm

The message receiver decompresses the name tag integers as follows.

When the received name tag integer is greater than or equal to the constant offset used to skip over name tag integers with global meanings such as the common end tag integer, the constant offset is subtracted from the received name tag integer and the result is used to index into the current name tag lookup table. The name tag character string at that entry is obtained and output with the appropriate XML punctuation.

If the name tag was for an attribute, the attribute value would be decoded back into a character string, if not one already, using the value type indicated by the name tag lookup table entry as a guide.

If, on the other hand, the name tag were for a child element, the element ancestry integer from the name tag lookup table entry would be pushed to the stack. The new element ancestry integer now on the top of the stack switches the current name tag lookup table to the one associated with the child element context for use by the name tag integers that follow.

When the received name tag integer is the common end name tag integer, the receiver obtains the end element name tag character string from the element ancestry lookup table at the index specified by the element ancestry integer at the top of the stack. It outputs the element end name tag character string with appropriate XML punctuation and then pops the stack.

Since the XML syntax places attributes before the “>” character defining the element begin to which they apply, the message receiver must delay output of the “>” character until detecting that all attributes of the current element begin have been received. This delay can be implemented by a local flag to indicate that the output of the “>” character is pending. The flag is set after an element begin name tag integer has been processed and cleared after a non-attribute name tag integer has been processed. It is tested when a non-attribute name tag integer is detected.

3.4.3 State Synchronization

So far, it has been assumed that the receiver’s stack is empty prior to the arrival of the root name tag of a new message. As stated above, a stack removes redundant information from the message. The following discussion considers the more general case in which the receiver connects to a message stream already in progress at an arbitrary point in the stream rather than at the beginning of a message. For the receiver to interpret properly the message stream content, it must synchronize its stack with the message stream transmitter.

When no stack is used and every element begin, attribute, and element end name tag is assigned a globally unique integer, the receiver knows immediately the name tag being received without depending on neighboring name tag integers. When a stack is introduced so that all element end name tags can use the same common end tag integer, the receiver skips any initial common end tag integers because their meanings are ambiguous.

Synchronization is more complex for the ancestry-dependent name tag integer case. Here, the interpretation of each name tag integer depends on its element ancestry, except for the root name tag integer, which has no ancestors. The inability to distinguish between root and non-root name tag integers makes matters even worse. A simple means of state containment is needed.

In practice, synchronization on any arbitrary name tag is seldom necessary. Synchronization in a message\stream application is normally required only at the message or “stanza” boundaries because incomplete messages have little value out of context and are often discarded. Typically, one or two name tags delineate these synchronization boundaries. The boundary name tags also tend to occur relatively infrequently in the message stream sequence compared to non-boundary name tags collectively.

The solution is to assign globally unique name tag integers to boundary name tags and to treat all other name tags in the ancestry-dependent manner described above. For example, the common end tag integer could be assigned the value zero as before, the message boundary name tag integer could be assigned the value one, and all the remaining ancestry-dependent name tag integers could be assigned values greater than or equal to two. The common offset used to relate name tag integers to name tag lookup table indices described above would have a value of two.

3.4.3.1 Name Tag Integer Atomicity

The synchronization approach assumes that the receiver can recognize the name tag integers as properly atomic. To illustrate the concern, assume that name tag integers in the message stream are represented by 2 bytes. The receiver may regain reception on either the first or second byte of an integer. If it has no means of knowing from which byte reception has been recovered, it has a 50-percent chance of assuming the wrong byte. XML name tags avoid this problem because their

text is enclosed between the “<” and “>” characters, which clearly delineate where the name tag begins and ends.

The TCP network protocol provides atomicity at the byte level. It does not guarantee that 2-byte alignment, or any multibyte alignment for that matter, will be preserved. It simply provides a “stream” of bytes. A simple solution is to require all name tag integers to fit within a single byte. The ancestry-dependent name tag integer approach greatly helps by keeping the range of name tag-integer values quite small.

Occasionally, name tag integers will not fit in a single byte. One solution is to replace the parent name tag integer with a set of parent name tag integers that all represent the same parent name tag. The various child name tags are then distributed among the members of the set of parent name tag integers. Unfortunately, the optimum distribution is not always clear. Furthermore, when a root name-tag integer is too big, a pseudo name tag integer must be created to act as its parent. These factors make this approach complex and undesirable.

A better approach is to use a binary encoding that distinguishes among the bytes comprising the name tag integer. This approach has the advantage that extra bytes are needed when only the most significant bytes represent non-zero bits of the integer being encoded. Both Unicode Transformation Format (UTF-8) and WBXML use this approach. They set the most significant bit of a byte to zero when the seven remaining bits of the byte represent the least significant bits of the integer. Otherwise, the most significant bit is set to one. Assuming that bytes are received in a big-endian sequence, as is the case for the TCP-network protocol, reception of a byte with its most significant bit set to zero could represent fully a single-byte integer, or it could be the least significant byte of a multibyte integer. The encoding is ambiguous. In either case, however, it is the last byte of that integer. The second received byte, in the context of the first received byte, is defined completely because it begins a new integer.

3.4.3.2 Message Atomicity

A software tool should be designed to generate the element ancestry lookup table and all the associated name tag lookup tables from an input XML schema file. Currently, the XML schema specification does not provide a means of specifying that a name tag indicates a synchronization boundary. The tool needs this information so it can assign the selected name tag a globally unique integer. Thus, that information, in addition to the XML schema file, must be included as an input to the tool.

3.4.4 State Syntax Checking

The transmitter can detect syntax errors when an element begin name tag or attribute name tag appears that is not recognized during the search of valid character strings in the current name tag table. This form of syntax error checking is a natural result of the search process and has no impact on the table structure.

The XML schema specifies not only what child elements and attributes are permitted for each parent elements, but also how the child elements may be sequenced and how many occurrences of each are permitted. Due to the potentially large number of valid permutations, adding syntax error checking for these additional restrictions requires a more complex state machine with potentially many more states. Its design is beyond the scope of this project. It is mentioned only to note that the XML name tag compression described here could be integrated into it.

3.4.5 User Data

The message receiver needs the ability to distinguish between a numeric value that represents a name tag integer and a numeric value that represents user data. Three approaches are considered below.

The first approach differentiates between name tags and user data by assigning them different binary representations. XML encodes user data using printable character strings. The binary representation of the non-printable characters can function as name tag integers. The approach is simple to implement and does not suffer from the synchronization problems of other approaches. Its primary drawback is that all user data are retained as a character string without compression.

The second approach is to use knowledge of the length and location of user data derived from the XML schema element type definitions or, in the case of variable-length data, specified directly in the message before its presentation. This approach allows numeric data to be encoded in binary format rather than in an equivalent character string format, which improves compression of numeric data significantly. The receiver differentiates between name tag integers and integers representing user data, not by the integer values, but by where the integers are located in the message. The implementation is considerably more complex than the approach described above because processing of user data types must be supported. The approach is also unsuitable for applications that involve attempts to access a message stream at an arbitrary point and synchronize at the first received message boundary. Knowledge of the length and location of user data requires prior synchronization to some reference point. Otherwise, the representation of user data can be mistaken for the same representation encoding a synchronization boundary.

The third approach is to use escape sequences. An escape sequence translates an input user data character that conflicts into a sequence of characters that do not conflict. In the C, C++, C#, and Java™ programming languages, for example, a quotation mark terminates a string literal. If the string itself contains a quotation mark, the quotation mark must be preceded by a backslash character so that the string is not terminated prematurely. The backslash character initiates the escape sequence. XML also uses escape sequences. For example, the three characters “&gt” replaces the “>” character when the “>” character is not terminating an XML element name tag. For the compression approach described here, the escape character can be implemented using another globally unique integer.

3.5 SMALL TEXT XML COMPRESSION

The small text compression encoding described above compresses 8-bit ASCII text to approximately two-thirds of its original size by replacing nearly all of its most frequently expected characters with 5-bit characters. Four character subsets are defined. Each subset reserves three of its 5-bit character codes to allow transition to one of the other subsets. The first two subsets implement the 26 uppercase and 26 lowercase alphabetic characters, respectively. The other two subsets implement the numeric digits, punctuation marks, and an escape mechanism to handle any 8-bit ASCII character that may not be covered explicitly in one of the subsets.

3.5.1 Name Tag Integer Subset

To combine the benefits of the small text compression encoding with the name tag integer compression algorithm described above, two requirements must be satisfied efficiently. First, a means of representing name tag integers is needed in the 5-bit character framework. Second, they must be distinguished from application text, which also may contain integers.

Since name tag integers are so prominent, the two non-alphabetic 5-bit character subsets should be redefined so that one of the subsets is dedicated exclusively to name tag integers. The other subset

implements the numeric digits, the most important punctuation marks, and the escape sequence for arbitrary 8-bit ASCII characters. By assigning the name tag integers their own 5-bit character subset, they can be distinguished from other text. Furthermore, the 5-bit character introducing the name tag integer subset can act as a delimiter to terminate other text that precedes it.

3.5.2 Name Tag Integer Separation

The next issue is how to encode name tag integers within their 5-bit character subset. Three requirements apply here. First, integers of any size need to be supported. Name tag integers generally are expected to have small numeric values. No artificial limits should be imposed, however, that would limit the universality of the approach. Second, small numeric values, being more frequent, should require fewer 5-bit characters to encode, compared to larger numeric values. Finally, since name tag integers often occur one immediately after another, a means is needed to determine where in the sequence of 5-bit characters one name tag integer ends and the adjacent one begins.

All the subsets implement the space character with the same 5-bit character code as the one that other subsets use to introduce the subset. This 5-bit character's double duty is possible because a subset does not need to be introduced when it is already in current use in that subset. One approach uses the space character within the name tag integer subset as the separator between adjacent name tag integers.

Another approach uses a scheme similar to what 8-bit UTF-8 implements for multibyte characters. It assigns a bit within each byte to indicate whether the next byte is part of the same character. To implement this approach in a 5-bit character subset, each name tag integer digit value would have two 5-bit character representations, one indicating that an additional digit follows that is part of the same name tag integer, and the other indicating that it is the
last digit of the name tag integer. No additional 5-bit character is needed to separate name tag integers because the 5-bit character for the last digit indicates the end of the previous name tag integer.

By far the most frequently occurring name tag integer is the common end tag integer used to terminate XML elements. The first approach requires every common end tag integer 5-bit character to be preceded by another 5-bit character to introduce the name tag integer subset or to separate the common end tag integer 5-bit character from another name tag integer. In both cases, the preceding 5-bit character has the same code value. Only its interpretation differs.

The second approach does not require a 5-bit character preceding the common end tag integer 5-bit character to separate it from a preceding name tag integer. Given the common end tag integer's high frequency of occurrence, it should be coded with as few 5-bit characters as possible. Therefore, the second approach was selected.

3.5.3 Name Tag Integer Digits

A 5-bit character subset supports 32 possible code values. Three of these code values are reserved for transitions to the other 5-bit character subsets. The end-of-message code, `<00000>`, is reserved to support the 5-bit character boundary synchronization algorithm described previously, which leave 28 possible code values available for the function that the 5-bit character subset implements. Of these, 26 form a continuous sequence of code values. They range from `<00001>` to `<11010>` inclusive.

Each digit requires two 5-bit character codes, one for when additional digits of the same name tag integer follow, and the other for when no digit of the same name tag integer follow. To support the 10 digits from 0 through 9 therefore requires 20 5-bit character codes.

If, instead of base-10 digits, base-13 digits were used, 26 available 5-bit character codes would be employed. The range of name tag integer values that a given number of 5-bit characters can represent is more extensive. One 5-bit character supports values up to 12 rather than 9, two 5-bit characters support values up to 168 rather than 99, three 5-bit characters support values up to 2,168 rather than 999, and so on. Base-13 digits provide better compression compared to base 10-digits because more information is represented in the same number of 5-bit character bits.

The base-13 digit that each associated 5-bit character represents, and whether additional 5-bit characters for base-13 digits are expected immediately after it, can be implemented easily by using a lookup table. Figure 10 provides an example of a large-value base-13 name tag integer computation.

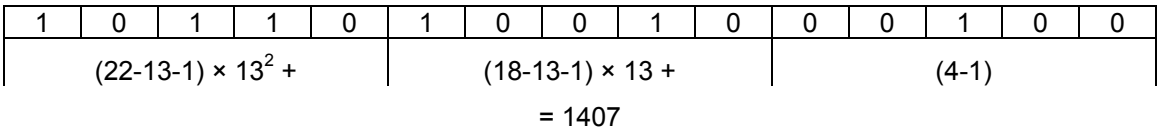

Figure 10. Large-value name tag integer example.

Base-13 digits employed 26 of the 28 available 5-bit character codes. One could use base-14 digits instead to employ all available 5-bit character codes. The decision not to do so was based on the desire to leave two 5-bit character codes available for possible future expansion and to simplify the algorithm performing base conversions.

4. BIBLIOGRAPHY

Apostolico, A., M. Comin, and L. Parida. 2004. "Motifs in Ziv-Lempel-Welch Clef," *Proceedings of the Data Compression Conference 2004*, pp. 72–81. Institute of Electrical and Electronics Engineers (IEEE) Computer Society. New York, NY. Presents variations of dictionary-based compression.

Deutsch, P. and J-L. Gailly. 1996. "ZLIB Compressed Data Format Specification Version 3.3," RFC-1950 (May). http://www.gzip.org/zlib/rfc1950.pdf.

Deutsch, P. 1996. "DEFLATE Compression Data Format Specification Version 1.3," RFC-1951 (May). http://www.faqs.org/ftp/rfc/rfc1951.pdf.

Dewes, C., A. Wichmann, and A. Feldmann. 2003. "An Analysis of Internet Chat Systems," *Proceedings of the 3rd SIGCOMM Conference on Internet Measurement*, pp. 51–64. Association for Computing Machinery (ACM) Press, New York, NY. Presents data on chat network traffic characteristics.

Girardor, M. and N. Sundaresan. 2000. "Millau: An Encoding Format for Efficient Representation and Exchange of XML over the Web," *Proceedings of the Ninth International World Wide Web Conference*, 15–19 May, Amsterdam, Netherlands. http://www9.org/w9cdrom/154/154.html.

Goldman, O. 2004. "Binary XML: Taking on XML's Bandwidth and Memory Constraints," *Dr. Dobb's Journal* (November), pp. 54–56. Presents brief overview of work in binary XML compression.

Hildebrand, J. and P. Saint-Andre. No date. "JEP-0138: Stream Compression (Draft)," XMPP Standards Foundation. http://www.jabber.org/jeps/jep-0138.html.

Holtz, K. "The Evolution of Lossless Data Compression Techniques," *WESTCON 1993 Conference Record*, pp. 140–145. Available through IEEE Digital Library (membership required).

Larsson, J. and A. Moffat.1999. "Offline Dictionary-Based Compression," *Proceedings of the Data Compression Conference 1999*, vol. 88, no. 11, pp. 296–305, IEEE, New York, NY. Available through IEEE Digital Library (membership required).

Martin, B. and J. Jano. 1999."Binary XML Content Format," W3C Note 24 June 1999, Wireless Application Forum Ltd.* http://www.w3c.org/TR/wbxml/.

Pereira, R. 1998. "IP Payload Compression using DEFLATE," RFC-2394, The Internet Society, Restion, VA. http://www.faqs.org/ftp/rfc/pdf/rfc2394.txt.pdf.

Piepgrass, D. No date. "Letter Frequency Counter," Web page: http://millikeys.sourceforge.net/freqanalysis.html. Presents statistics on English single-character and digraph frequencies.

Serin, E. 2003. "Design and Test of the Cross Format Schema Protocol (XFSP) for Network Virtual Environments," Master's Thesis, Naval Postgraduate School. Monterey, CA. http://theses.nps.navy.mil/03Mar_Serin.pdf. Presents the concept of replacing XML tags with numbers.

* now Open Mobile Alliance

Sundaresan, N. and R. Moussa. 2001. “Algorithms and Programming Models for Efficient Representation of XML for Internet Applications,” *Proceedings of the 10th International Conference on the World Wide Web 2001*, pp. 366–375. Association for Computing Machinery, New York, NY.

Veeneman, D. No date. Web page: http://www.etoan.com. Presents statistics on English single character and digraph frequencies.

Yergeau, P. 1998. “UTF-8: A Transformation Format of ISO 10646,” RFC-2279, The Internet Society, Reston, VA. http://www.ietf.org/rfc/rfc2279.txt.